A Study of Initialization in Linux and OpenBSD

Catherine Dodge, Cynthia Irvine, Thuy Nguyen
cdodge@alum.wellesley.edu, {irvine,tdnguyen}@nps.edu
Naval Postgraduate School

February 15, 2005

Abstract

The code that initializes a system can be notoriously difficult to understand. In secure systems, initialization is critical for establishing a starting state that is secure. This paper explores two architectures used for bringing an operating system to its initial state, once the operating system gains control from the boot loader. Specifically, the ways in which the OpenBSD and Linux operating systems handle initialization are dissected.

1 Introduction

Despite an extensive body of research regarding operating system development, there is little information about operating system boot and initialization. This situation may have two causes. First, initialization may be unimportant. Alternatively, how a system is initialized could be vitally important but simply not well understood. This paper explores the architectures used for initialization in order to better understand its impact within the operating system. The objective of this paper is to contribute to a larger effort [1] to better understand how initialization does or does not affect system security.

Two open source operating systems were chosen for study: OpenBSD and Linux. OpenBSD was studied in detail, while Linux initialization was explored to provide contrast. Although the study of a high assurance system, in the context of assurance as described in the Common Criteria [2], also would have been useful, there are currently no open implementations.

For exploring OpenBSD and Linux, the focus of study was the overall design and sequencing of their initialization routines. While the source code served as the primary focus of study, the intent was not to pursue a complete source code audit seeking actual vulnerabilities. Rather the focus was on overall design characteristics. In keeping with this methodology, comments in the code were for the most part taken at face value.

In the sections that follow, an overview of the entire boot process on a typical Intel x86-based PC is presented. Next we present an analysis of the initialization routines of OpenBSD and Linux, respectively. The paper ends with conclusions and directions for future work.

2 Overview of the PC Boot Process

We begin with an overview of the boot process of an Intel x86-based personal computer to illustrate how the initialization routine provided by the operating system fits into a larger picture. Generally, there are three stages to the bootstrapping process. When a PC is powered on, the BIOS (Basic Input-Output System) runs first, followed by a boot loader and finally the operating system initialization routine. The logical execution sequence of these components is shown in Figure 1.

Report Documentation Page					*Form Approved* *OMB No. 0704-0188*
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE **15 FEB 2005**		2. REPORT TYPE **N/A**			3. DATES COVERED **-**
4. TITLE AND SUBTITLE **A Study of Initialization in Linux and OpenBSD**					5a. CONTRACT NUMBER
					5b. GRANT NUMBER
					5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)					5d. PROJECT NUMBER
					5e. TASK NUMBER
					5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School 833 Dyer Rd., Code CS/Cp Monterey, CA 93943-5118**					8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)					10. SPONSOR/MONITOR'S ACRONYM(S)
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release, distribution unlimited**					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**15**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

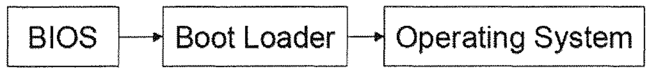

Figure 1. Logical Execution Sequence from Boot to Runtime on an Intel-based Personal Computer

2.1 The BIOS

The BIOS is the first code executed by the processor upon boot. When power is initially applied to the computer this triggers the RESET pin on the processor. This causes the processor to read from memory location 0xFFFFFFF0 and begin executing the code located there. This address is mapped to the Read-Only Memory (ROM) containing the BIOS. The BIOS must poll the hardware and set up an environment capable of booting the operating system. BIOS functionality can be broken into three areas: *Power On Self Test* (POST), *Setup* and *Boot*.

The main function of the POST is to test and initialize the various hardware components detected by the BIOS. As part of the POST routine, hardware devices also register themselves with the BIOS as associated with or "hooked" into certain interrupt values. For instance, by convention on the Intel platform, all hard drives register with the 13h interrupt. This allows programs to invoke this software interrupt to communicate with disk drives, utilizing device driver code within the BIOS. Note that while older operating systems such as DOS rely heavily on the BIOS to provide an interface to the hardware, newer operating systems are moving away from using BIOS routines to communicate with hardware. Many modern operating systems supply their own 32-bit device drivers once fully running, but these operating systems do make use of BIOS routines to initially load the operating system from disk.

Once POST operations are complete, the BIOS offers the user the option of entering "Setup" mode. It is here that the user may change the configuration of the BIOS, including the boot sequence.

The last action of the BIOS is to execute the 19h interrupt, which loads the first sector of the first boot device. Since this is the location of the boot loader, execution of the 19h interrupt transfers control to the boot loader. The hard drive and other devices, such as CD-ROM drives, can register as bootable and be put on a list of bootable devices, which is prioritized by the user during the "Setup" mode.

2.2 The Boot Loader

Once the BIOS loads the first sector of the boot device into RAM, the boot loader begins execution. In the case of a hard drive, this first sector is referred to as the *Master Boot Record* (MBR). The MBR contains the partition table describing the partitions defined on the hard drive. It also contains a program, the boot loader, which will load the first sector of the partition marked as active into RAM and execute it. The size of the MBR is limited to one sector on disk or 512 bytes, since it is located within the first sector of the drive at cylinder 0, head 0, sector 1. This size limitation greatly limits its functionality. Typically boot loaders have been highly integrated with the operating system that they support. This integration cuts down on the operations a boot loader must perform, making a 512 byte boot loader feasible. When more functionality is required, a multi-stage boot loader may be used.

A *multi-stage boot loader* provides more function and flexibility by working around the 512 byte size limitation. Rather than consisting of a single program which loads the operating system directly, multi-stage boot loaders divide their functionality into a number of smaller programs that each successively load one another. This architecture allows a fairly primitive boot loader, located in the MBR, to load and execute the next stage of the boot loader, a larger and more sophisticated boot loader. Subsequent stages can be located elsewhere on the hard drive and thus are not subject to the single sector size limit. This chaining of boot loaders allows the boot loader functionality to become arbitrarily complex. LILO [3] and GRUB [4] are two well-known multi-stage boot loaders capable of booting a number of operating systems including Linux, Windows and FreeBSD. Both first "bootstrap" themselves into operation, then present the user with a number of OS boot options.

Boot loader developers have access to the OS source code for open source operating systems, which allows them to understand and emulate how any of these operating systems boot. For proprietary operating systems however, developers of programs like GRUB have an alternate solution: the third party boot loader can be instructed to invoke the boot loader of the proprietary operating system, which then loads the proprietary OS. This type of *chain loading*, where one boot loader loads and executes another boot loader, leads to a clean and effective implementation. The pre-existing code for booting the OS is reused, a hallmark of good software engineering practice.

While boot loaders can exist as standalone software, as GRUB demonstrates, they are most often tailored to and integrated into a specific operating system. This has led to a situation where each operating system requires a customized boot loader. One movement to define a clear-cut API between the boot loader and operating system is the Multiboot Specification, developed and maintained by the Free Software Foundation [5]. The Multiboot standard defines three main aspects of the interaction between the boot loader and the operating system:

1. The format of the operating system image, as perceived by the boot loader,
2. The state of the system when the boot loader starts executing the operating system, and
3. The format of the information passed by the boot loader to the operating system

The Multiboot specification describes the header that must be included in the operating system image, the values that must be in certain registers (not all registers are specified), and the format of the information passed to the OS. Currently the only programs that fully support the specification are GRUB, the Mach kernel [6] and the Fiasco [7] operating system, while others such as Linux are somewhat Multiboot compliant.

The advantages of such a specification are several. It allows the developers of new operating systems to make use of existing boot loaders without having to write one themselves. Having such a clearly defined interface also adds a level of modularity and transparency to the boot process.

2.3 OS Initialization

Once the boot loader has loaded the OS image into memory, control is transferred to the OS. The operating system will go through a series of steps to set up the operating environment necessary to achieve a coherent initial run state. This "initialization" sequence executes once. Then the operating system enters its main scheduling loop.

A number of initialization tasks are common across operating systems. In particular, data structures needed for kernel operation must be defined before they can be used. Examples are the run queues used to manage scheduling and the structures used to represent files, *vnodes* in the case of OpenBSD. Many of the required initialization operations have interlocking dependencies. While some device drivers may need to create kernel space processes as part of their initialization, support for forking new processes may not yet be available. Thus the design of initialization code requires an acute awareness of interrelated dependencies.

A large aspect of initialization for any operating system is the establishment of virtual memory management. On an Intel-based system this typically involves setting up the Global Descriptor Table (GDT), creating a Local Descriptor Table (LDT), switching the processor into protected memory mode, setting up page directories and enabling paging.

Additional tasks include device driver initialization and the assignment of interrupts in the Interrupt Descriptor Table (IDT). Another major initialization task is establishing support for various file system types and mounting a root file system. The ultimate goal of many of these initialization tasks is the establishment of support for multi-tasking, the central purpose of an operating system.

The notion of a *process* is the basic unit of measuring execution in the various versions of UNIX. Throughout this paper the term *process* is used in this UNIX sense. Depending on the implementation, each process may or may not have its own Task State Segment (TSS), defining it as a *task* in the Intel vocabulary [8]. One of the conundrums of the initialization sequence is that the code that first executes in the kernel must be able to turn itself into a process that can be swapped in and out of a running state. This is how the appearance of multi-tasking is achieved. Every other process is created during runtime by using kernel functions designed to support process creation, e.g. a *fork()* call. Yet the initial process must explicitly do for itself all the tasks accomplished by a call to *fork()*. This initial process, numbered 0 on UNIX systems, will be referred to as Process 0 in this discussion. Process 0 must be able to self-generate its own process context. Once this context has been established, the system has the capability to suspend and resume execution of Process 0 just as it would any other process. Once established, the role of Process 0 differs by operating system. These behaviors are discussed more fully in subsequent sections.

While Process 0 is responsible for the bulk of kernel data structure generation on UNIX systems, it is Process 1 that is most familiar to users. Process 1, commonly referred to as the *init* process, is the first process forked from Process 0. To avoid confusion, the term "init process" will be avoided in favor of the term Process 1. It is Process 1 that handles user space initialization. The term "user space" refers to code running in user mode, which on an x86 system means code running within privilege level 3. In contrast, Process 0 runs entirely in kernel space, e.g. on an x86 system this means running within privilege level 0. Depending on the operating system, Process 1 may perform a few tasks before doing its most important job – executing the *init* binary that will handle user space tasks such as starting application daemons and setting network configurations. This analysis will be limited to kernel space initialization, thus Process 1 actions within user space will not be considered. Once Process 1 has been forked from Process 0, often a number of additional kernel space processes are created to handle additional kernel space tasks. Once all of these are running, the kernel space operating system initialization is complete.

3 OpenBSD Initialization

Although similar to other versions of UNIX, the OpenBSD project has focused a great deal of effort on security. This has largely taken the form of ongoing source code auditing [9]. In the following discussion, initialization of the system will be traced from the boot loader through the kernel code responsible for bringing up the system to the point where the scheduling loop is entered.

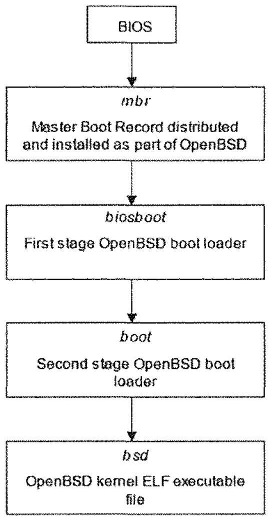
BIOS
mbr
Master Boot Record distributed and installed as part of OpenBSD
biosboot
First stage OpenBSD boot loader
boot
Second stage OpenBSD boot loader
bsd
OpenBSD kernel ELF executable file

Figure 2. OpenBSD Logical Execution Sequence from Boot to Runtime

Release 3.4 of the OpenBSD operating system was used for the analysis presented here and the path of all files referenced is rooted at the */src/sys/* directory within the OpenBSD source tree. Code that requires a compile-time flag to be included in the executable has been omitted from this discussion. First an outline of the boot process is necessary, which will be followed by a more detailed explanation of the initialization tasks performed by the kernel. Figure 2 shows the logical execution sequence during the OpenBSD boot process.

3.1 Boot Loader Initialization

OpenBSD employs a two-stage boot loading process, if one does not count the MBR as a separate "stage." The first stage is handled by a boot loader program called *biosboot*, while the secondary boot loader is called simply *boot*. An MBR image is also included with the OpenBSD distribution. In the past, the *biosboot* program was used as the MBR itself, but the *biosboot man* page no longer recommends this due to strange interactions based on BIOS peculiarities. At boot time, control moves from the BIOS to the MBR then to the initial boot loader, *biosboot*. This program is installed using a special tool called *installboot*, which patches *biosboot* with information about the location of the second stage boot loader, *boot*. Thus if the location of *boot* were changed, *biosboot* would no longer be able to find the secondary boot loader and the boot process would fail. Once *biosboot* has successfully located and loaded the *boot* program from disk into memory, it transfers control to this newly loaded program.

The *boot* program sets up an environment suitable for transferring control to the kernel image. It also provides an interactive prompt for user input of additional boot parameters. The main tasks of the *boot* program are:

1. Switching the CPU into protected mode
2. Probing for console devices and displaying subsequent messages to the discovered consoles
3. Detecting memory, both that reported by the BIOS and extended memory
4. Detecting if the BIOS supports Advanced Power Management (APM)

The program then reads from */etc/boot.conf* any specified boot parameters that will be used to locate and boot the kernel. Next the user is prompted for any interactively passed boot parameters. If none are entered, then after the timeout period, the kernel at the default location */bsd* will be loaded using the parameters found in */etc/boot.conf*. Once the kernel has been loaded, *boot* is finished and control of system initialization is transferred to the kernel executable file, *bsd*.

3.2 Kernel Initialization

Kernel initialization, which is comprised of both initialization requiring assembly level control and high level initialization will be discussed in this section.

3.2.1 Assembly Level Initialization

Upon a successful load of the kernel, execution begins at the *start* label found in the assembly code file */arch/i386/i386/locore.S*. Figure 3 shows the flow of control within the OpenBSD kernel during the initialization phase. The code in *locore.S* loads the parameters passed to it by the boot program via the stack. Thus, while there is interaction between the boot program and the kernel, it is handled in a manner similar to a function call. Parameters to be passed are put on the stack by *boot*. It is assumed that the kernel knows how to find these stack-located parameters.

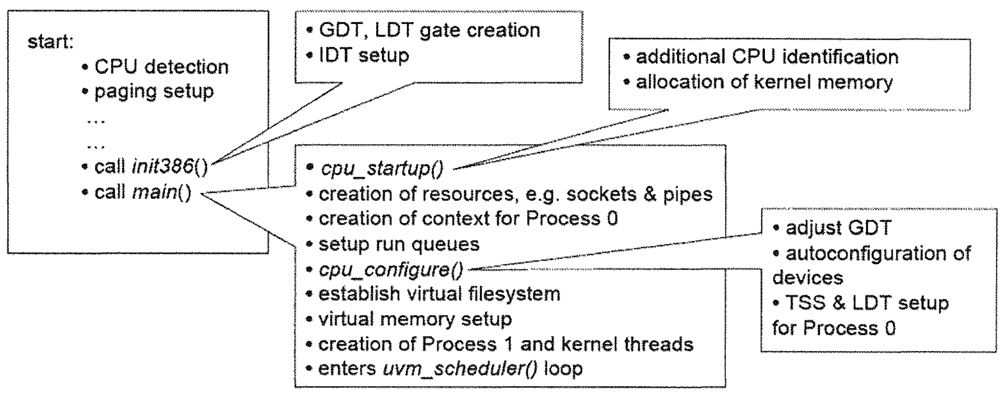
start:
• CPU detection
• paging setup
...
...
• call *init386*()
• call *main*()
• GDT, LDT gate creation
• IDT setup
• additional CPU identification
• allocation of kernel memory
• *cpu_startup()*
• creation of resources, e.g. sockets & pipes
• creation of context for Process 0
• setup run queues
• *cpu_configure()*
• establish virtual filesystem
• virtual memory setup
• creation of Process 1 and kernel threads
• enters *uvm_scheduler()* loop
• adjust GDT
• autoconfiguration of devices
• TSS & LDT setup for Process 0

Figure 3. Execution Path within *bsd*, the OpenBSD Kernel Executable, During Initialization

Next the code determines the exact type of x86-compatible processor the kernel is running on. This information is needed in order to handle processors that do not implement some instructions in hardware and thus require software emulation. A page table directory is subsequently constructed and paging is enabled. After this is finished, *init386()* is called.

The function *init386()* is responsible for initializing many of the x86-specific structures, particularly those related to memory management. This includes creating the GDT, LDT, GDT memory segments, and LDT memory segments, as well as initializing the IDT. The function *pmap_bootstrap()* is then called, which does a first pass of initializing the physical mapping (pmap) module of the memory management. The function *pmap_init()* finishes initializing the pmap module later, after control enters the *main()* routine[1]. Finished with these tasks, *init386()* returns control to the assembly file *locore.S*. At this point, a call is made to *main()*, the machine-independent C language function defined in *init_main.c*.

3.2.2 High-level Initialization

The *main()* function calls all of the various subsystem-specific initialization routines needed to prepare the operating system to run. The function is completely architecture independent, thus *main()* calls only high-level initialization functions. It is only in the implementation of functions called by *main()* that machine dependent features are encountered. The *main()* function is described below in a strictly linear fashion, but to aid in understanding, the work done within *main()* can be roughly grouped into three categories.

3.2.2.1 Initialization of Various Subsystems

Initialization of the current process pointer is the first step. The kernel process that currently has control is defined to be Process 0. The second step is to initialize the console so that any debugging messages can immediately be written to the screen. Next kernel data structures used for

[1] The pmap module provides translation of high-level memory management functions to the appropriate machine dependent functions. Thus the pmap module implements, for example, architecture-specific values for the three permission levels—read, write and execute—defined by the machine-independent memory protection. Once this basic level of memory support has been established, but before the memory management unit (MMU) has been fully initialized, it is possible for memory to be dynamically allocated via the *pmap_bootstrap_alloc()* call. Memory allocated in this way will not be managed by the MMU but instead can be considered a hard-wired mapping. Once *pmap_init()* is called and the MMU has been fully initialized, memory can be allocated normally using MMU support.

autoconfiguration, virtual memory, disk management and TTY management are initialized. The following function called, *cpu_startup()*, handles most of the architecture-specific initialization.

Within *cpu_startup()* the buffer used for writing error messages is first initialized, using memory reserved when *pmap_bootstrap()* ran. Next the CPU is identified, which is needed at this point in order to be able to handle processor-specific bugs, such as the "f00f" bug found in older Intel processors [10]. The *allocsys()* function that follows is responsible for allocating memory for kernel data structures. Memory is allocated for the entire kernel, to *kernel_map*, and subsequently assigned to specific submaps that were initially defined when *uvm_init()* was called earlier from within *main()* to initialize the structures for virtual memory support. *Submaps* are subsections of kernel memory that are defined in order to isolate the memory usage of various subsystems from one another. Submap usage also subdivides the locking of kernel memory into smaller pieces. Once memory is assigned to the various maps, *cpu_startup()* returns control to *main()*.

The next step in *main()* is to call the function *mbinit()* which initializes a pool of *mbuf* data structures. The *mbuf,* or memory buffer structure, is the foundation of the memory management used to handle networking. The function *soinit()* is subsequently called, which reserves memory space for use by sockets by creating a resource pool named *socket_pool*. The next few high level functions initialize the data structures and resource pools to support timeouts, sysctls, process management and file descriptors. The call to *pipe_init()* similarly sets up a resource pool to support pipe structures.

3.2.2.2 Establishment of Process 0 Context

To correctly establish Process 0, data must be inserted into various lists. While many resources and structures needed to support Process 0 were set up in *locore.S*, at that time structures such as the process queues did not exist. Thus it is only now that, for example, an entry for Process 0 can be added to the process ID hash table. Other items that are defined to create the context for Process 0 include the *p_stat* (set to *SRUN*) and *p_nice* (set to 20) values. Finally Process 0 is given the name of "swapper."

Much of the additional context to be defined for Process 0 involves setting variables within the *proc* structure for Process 0. All of the variables named in this paragraph are defined within the *proc* structure and initialized for Process 0 in the following order:

1. A timeout, named *p_sleep_to*, is established that will timeout when this process returns from calls to any of the *sleep()* family of functions which provide for voluntary context switching.
2. A second alarm timeout, named *p_realit_to*, is created which allows processes to base actions on a "real" timer that decrements in real time.
3. The credentials structure, the *pcred* structure *p_cred,* is initialized. This is used to store the real and effective user id and group id values used to determine permissions.
4. The resource pool used for allocating all signal structures is initialized by calling *siginit()*.
5. A new signal structure is created and the signals to be ignored are established by setting the *p_sigignore* value, a *sigset_t* structure.
6. A file descriptor table is built by calling *fdinit()* and assigning the returned file descriptor table to the *p_fd* element of the *proc* structure.
7. Limits on nine different resources, such as number of open files, that can be controlled on a per process basis are established within the *p_limit*, a *plimit* structure.

The next section of code within *main()* defines the address space map for Process 0. First *uvmspace_init()* is called which initializes the *vmspace* structure *vmspace0*. The address space for Process 0 is then defined to be this *vmspace0* structure. Last, two additional values within the *proc* structure are assigned, *p_addr* and *p_stats*. At this point enough of the supporting data structures are in place to be able to assign this process to *root*, i.e. the number of processes listed as running under user *root* is incremented to one.

3.2.2.3 Additional Subsystem Configuration

The run queues, 32 in total, used by the scheduler are zeroed via a call to *rqinit()*. At this point we reach a call to a fairly complex function, *cpu_configure()*. Some of the most important tasks handled by *cpu_configure()* include:

1. Adjusting the GDT to an appropriate size via *gdt_init()*
2. Autoconfiguration of all devices via a call to *config_rootfound()*
3. Initialization of the TSS and LDT for Process 0 by calling *i386_proc0_tss_ldt_init()*

After *cpu_configure()* returns to *main()*, the next function call, *uvm_init_limits()*, establishes limits for Process 0 on virtual memory usage. This function is only called once in the kernel code, as every other process will inherit these limits from Process 0. Support for the virtual file system is then established via a call to *vfsinit()*. This function sets up the *vnode* management structures and prepares the system able to handle any type of file system. The *vnode* structure is the basis of file management in OpenBSD.

The next call in *main()* is to *initclocks()* to establish the real time and statistical clocks. Next in *main()* attaches all the pseudo devices by first ensuring that the random device is attached, then it calls the attach function for each pseudo device.

Initialization of the various networking protocols supported by the system is handled by the following section of *main()* code. The initialization routines are bracketed by calls that raise the interrupt priority level to that of *IPL_VM*, and then return the priority level to its prior value. This effectively blocks the receipt of network traffic until the proper subsystems are initialized. The intervening three function calls set up a timer needed for minding the network interface, define various communication domains for use by the system, and associate the domains with the various interfaces on the system.

The next function call, *init_exec()*, defines the maximum executable header size that will be possible for the system. These are based on formats such as ELF (Executable and Linkable Format), COFF (Common Object File Format) and a.out. At this point, the timers used for scheduling are enabled within *scheduler_start()*. The following function called from *main()*, *dostartuphooks()*, executes all functions found in the *startuphook_list* queue. The routine *startuphook_establish()* can be used to add a function to this queue which will then be run at startup. This feature can be useful for certain device drivers.

The next steps configure the root and swap devices, by running checksum algorithms over them and assigning major and minor device numbers. Once configured, the root device is mounted via a call to *vfs_mountroot()*. The *vnode* structure allocated for the root directory is then retrieved and the current directory of this process is set to this *vnode* value. Upon completion of this file system management, Process 0 completes initialization of the virtual memory subsystem by calling *uvm_swap_init()*. This function initializes the swap system structures, thus it had to be called after file system initialization since it relies on the ability to work with *vnodes* from the recently mounted file system.

Some needed accounting takes place at this point. The start time of Process 0 is updated to the current time and the length of time that Process 0 has been running, the *p_rtime* value within the *proc* structure, is reset to zero.

3.2.2.4 Creation of Additional Processes

The creation of other processes may begin. It is essential that Process 0 be properly established before attempts to create other processes are made. When a new process is created, the *fork* call copies much of the information needed by the new process from the process structure of its parent process. Incorrectly setting the parameters of Process 0 would result in the propagation of erroneous context information to all child processes.

Two kinds of child processes are created by Process 0: the Process 1 and kernel processes. To create Process 1, which will run the *init* program, a call to *fork1()* is made. To create the other six kernel processes needed, a call to *kthread_create()* is used. This latter function is basically a specialized version of *fork1()*, as delving into the implementation reveals that a call to *fork1()* is made within *kthread_create()*. Similarly, the fork system call is implemented via a call to *fork1()*. As a kernel function, *fork1()* can be called directly only by kernel code; it is called indirectly from user space using the *fork* system call. The reason for creating Process 1 differently becomes clear when one delves into the implementation of *kthread_create()*. Since it will become a user space process, Process 1 should share neither memory nor signal structures with Process 0. Therefore it cannot be created by a call to *kthread_create()* which shares each of those structures by default. Instead a custom call to *fork1()* is required to properly initialize Process 1. The processes created using the *kthread_create()* call include:

1. A "pagedaemon" process to handle page swapping for the virtual memory subsystem,
2. A "reaper" process to free the resources still allocated to dead processes,
3. A "cleaner" process to clear out dirty buffers found in the BQ_DIRTY buffer queue,
4. An "update" process for synchronizing the file systems,
5. An "aiodoned" process for handling completed asynchronous I/O operations,

A final few additional kernel threads are created by the call to *kthread_run_deferred_queue()*. This function creates additional kernel threads as specified in the *deferred request queue*. During initialization, when full support for forking processes was not yet in place, driver or file system code that needs to create a kernel thread can add a request to this queue via a call to *kthread_create_deferred()*. When *kthread_run_deferred_queue()* is called, that function processes the deferred queue, creating each of the requested kernel threads. The hierarchy of processes created on a typical OpenBSD system is shown in Figure 4.

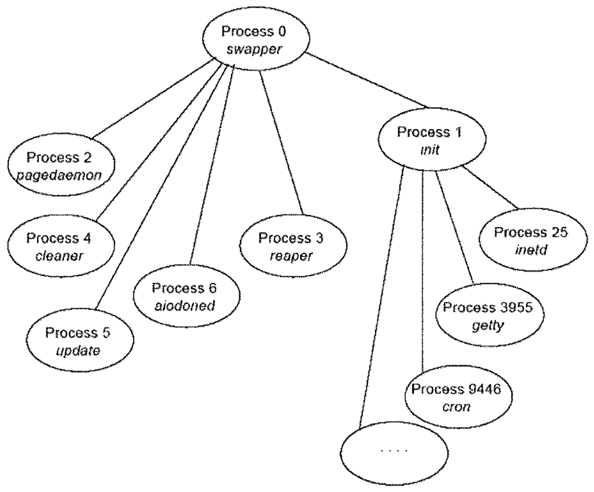

Figure 4. Process Hierarchy within OpenBSD

With all kernel threads running, only a few finishing touches are needed. The random number generator is seeded and the generation of process identification numbers is set up such that each successive process will be given a larger pseudo-random number than its predecessor. At this point, Process 0 finally enters its main loop by calling *uvm_scheduler()*. This function has Process 0 continually check for processes that are in a runnable state but not resident in memory and swaps them in. Control never returns to the *main()* function from this call and one could say the operating system is truly running.

It is important to note that Process 1 will go on to complete its initialization tasks after being forked from *main()* by Process 0 via a call to *fork1()*. Control for Process 1 moves first to the function *start_init()*, after being forked. This function checks to make sure a console device has already been properly initialized. Then it tries to execute each of the programs found on the *initpaths* list in turn. This list contains the potential candidates for an *init* program. Typically, a valid *init* program can be found at */sbin/init*. If this fails, two other possibilities are contained in the *initpaths* list. When a valid binary is found, this binary is executed by Process 1 as the *init* program. It will handle the remaining initialization required within user space. With all necessary processes now running, initialization is complete.

4 Linux Initialization

Linux 2.4.26 will be used in our overview of Linux kernel initialization. All references to source code file paths are relative to the Linux root source directory, typically named *linux-[version number]*. Much of this discussion relies on a text by Bovet and Cesati [11]. Figure 5, provides an overview of the execution sequence during the boot process of the Linux kernel.

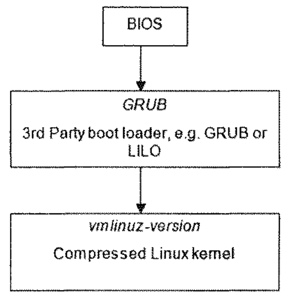

Figure 5. Linux Logical Execution Sequence from Boot to Runtime

4.1 Assembly Level Kernel Initialization

After the chosen boot loader has run, it loads the Linux kernel image, typically named *vmlinuz-[version number]* for a compressed kernel image and *vmlinux-[version number]* for an uncompressed image. A compressed kernel image will have the Linux boot loader, found in */arch/i386/boot/bootsect.S*, located at the very beginning of the image. An uncompressed kernel image is simply an ELF executable [12]. The discussion below traces the execution flow for a compressed kernel image.

After the boot loader provided runs, execution begins at the *start* label in the assembly code file */arch/i386/boot/setup.S*. Some of the operations performed by this code include:

1. Reinitializes all hardware, since Linux does not rely on the BIOS to do this properly.
2. Ensures interrupts are disabled
3. Sets up a provisional GDT and a provisional IDT
4. Reprograms the Programmable Interrupt Controller (PIC)
5. Re-maps the 16 IRQ lines from 0 thru 15, the BIOS assigned values, to 32 thru 47.
6. Switches from real mode to protected mode memory addressing.

The last line executed in this file is a jump to an assembly function called *startup_32()*, which performs additional initialization. There are actually two *startup_32()* assembly functions in the Linux kernel, located at */arch/i386/boot/compressed/head.S* and */arch/i386/kernel/head.S*. The reason that these duplicate names do not cause any naming conflicts is because each function is reached by jumping to a physical address, rather than via a call to a label. The existence of both versions is an artifact of when there was a single *head.S* file, before compression was implemented for the kernel image. At the end of the initial assembly code in */arch/i386/boot/setup.S* a jump to offset 0x100000 in segment __KERNEL_CS is called. This is where the version of *startup_32()* found in */arch/i386/boot/compressed/head.S* is located. But as part of the decompression routine, the entry point of the decompressed kernel, *startup_32()* in */arch/i386/kernel/head.S,* is relocated to that same address, 0x100000. Thus at the end of *startup_32()* found in */arch/i386/compressed/head.S* a jump to 0x100000 is called to reach the *startup_32()* found in */arch/i386/boot/head.S.* Thus while it appears from the code that these two jumps to the same physical location 0x100000 would be repetitive, they in fact are not.

The first instance of *startup_32(),* found in */arch/i386/boot/compressed/head.S,* performs the following:

1. Initializes the segmentation registers
2. Sets up a provisional stack
3. Decompresses the kernel image and locates it at address 0x10000
4. Jumps to the *startup_32()* function in the decompressed kernel, in */arch/i386/boot/head.S.*

The second *startup_32()*, located in */arch/i386/kernel/head.S*, continues the initialization sequence. Figure 6 details the flow of execution from this second *startup_32()* function onward through initialization, highlighting important tasks. This latter function's main job is to set up an environment within which the first process can execute. This includes:

1. Initializes the segmentation registers with their final values.
2. Sets up the Kernel Mode stack for Process 0.
3. Initializes the provisional kernel Page Tables
4. Stores the address of the Page Global Directory in the cr3 register, and enables paging by setting the PG bit in the cr0 register.
5. Fills the bss segment of the kernel with zeros.
6. Invokes *setup_idt()* to fill the IDT with null interrupt handlers.
7. The first page frame is loaded with the system parameters learned from the BIOS and the parameters passed to the operating system from the boot loader.
8. Loads the gdtr and idtr registers with the addresses of the GDT and IDT tables.

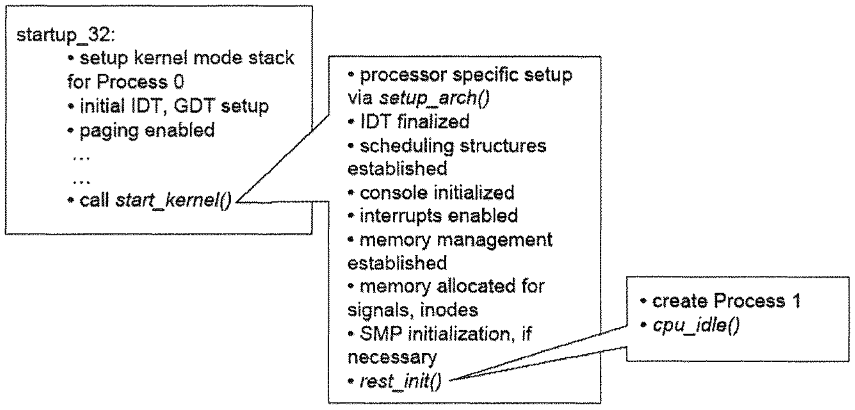
startup_32:
• setup kernel mode stack for Process 0
• initial IDT, GDT setup
• paging enabled
...
...
• call *start_kernel()*
• processor specific setup via *setup_arch()*
• IDT finalized
• scheduling structures established
• console initialized
• interrupts enabled
• memory management established
• memory allocated for signals, inodes
• SMP initialization, if necessary
• *rest_init()*
• create Process 1
• *cpu_idle()*

Figure 6. Execution Path within *vmlinux-[version]*, the uncompressed Linux ELF Executable, During Initialization

4.2 High-level Kernel Initialization

After completing these tasks, control jumps to the *start_kernel()* function. This function's role is the final initialization of all kernel components and structures. Since it touches on nearly all structures, it is instructive to discuss the important functions that *start_kernel()*, defined in */init/main.c*, performs.

- The processor specific initialization details are performed via *setup_arch()*, such as initialization of page tables and descriptors using *paging_init()*.
- The final initialization of the IDT is performed by invoking *trap_init()* and *init_IRQ()*.
- Data structures needed by the kernel for scheduling are created by *sched_init()*.
- Two of the four softirq types used for deferrable kernel functions are registered in the softirq array by invoking *softirq_init()*.
- The time and date used by the system are initialized in *time_init()*.

- The console device is initialized here before PCI handling is enabled. This ensures error messages will be reported to the console as soon as possible.
- The slab allocator is initialized by *kmem_cache_init()* and *kmem_cache_sizes_init()*.

Interrupts are enabled via a call to *sti()*, a wrapper for the assembly level instruction *sti* that sets the IF flag in the EFLAGS register. With this bit set, "the processor begins responding to external, maskable interrupts after the next instruction is executed." [13]. This is done after *trap_init()* and *init_IRQ()* have been called to fill the IDT. While any person familiar with assembly would recognize that interrupts have been enabled, much of the enabling and disabling of interrupts found in the rest of the kernel becomes quite complex.

Linux provides a large number of macros that serve as wrappers for the *cli* (interrupt disable) and *sti* (interrupt enable) instructions. What makes this even more complex is that these macros are then used inside of other macros, many layers deep, such that to track down the fact that a macro disables interrupts can require tracking through four levels of macro definitions. With interrupts enabled, execution resumes in *start_kernel()*, where a call is made to *calibrate_delay()*. It calculates an estimate of how many times per second a short loop can be executed by the CPU. Mainly used by device drivers, it provides a way for a driver to estimate how long it should busy-wait when waiting on information. The final few functions take care of the following tasks:

- The page frames are readied for use by the *mem_init()* function, which resets the reserved flag and calls *free_page()* on each page.
- The function *pgtable_cache_init()* initializes the x86 Physical Address Extension (PAE) caches on Intel processors with this feature
- The maximum number of processes allowed on the system is set to a default value in *fork_init()*
- Slab caches to support signals, the file system and memory are created by *proc_caches_init()*
- The slab caches as well as other kernel structures, such as inodes, needed to support the virtual file system are created by invoking *vfs_caches_init()*
- The buffer cache hash table for the file system is allocated in *buffer_init()*
- The page cache hash table is initialized in *page_cache_init()*
- The function *check_bugs()* identifies the CPU type and makes any necessary adjustments to compensate for known bugs in the respective chip
- If the kernel has been built for a uniprocessor system with an IO APIC, *smp_init()* initializes the APIC hardware. For a multi-processor system, this function is responsible for setting the system up to use all processors.

The final function called from within *start_kernel()* by Process 0 is *rest_init()*. This function invokes *kernel_thread()* to create the kernel thread that will become Process 1. A "kernel thread" in Linux is essentially a process that remains in kernel space, with access to kernel data structures. While execution for Process 0 remains inside the function *rest_init()*, the kernel thread just created, Process 1, proceeds to first execute the *init()* function found in the */init/main.c* file (only later will it execute the *init* binary which runs in user space). Discussion of Process 0 will resume later in this section. The following paragraphs trace the execution of Process 1.

The *init()* function that Process 1 enters performs a number of important final initialization tasks. First *lock_kernel()*ensures that none of the other kernel threads interfere with the device initialization that follows. The next function called is *do_basic_setup()*, which essentially handles all device initialization. A few details are in order. Within *do_basic_setup()*, the function *start_context_thread()* is called, which creates a new kernel thread. This new kernel thread, *keventd*, is responsible for running the tasks that accumulate in the *tq_context* task queue, which provides for deferred scheduling of functions that make use of blocking operations[2]. Figure 7 shows the process hierarchy of all kernel threads as well as a handful of well-known user space processes such as *sshd* and *mingetty*[3].

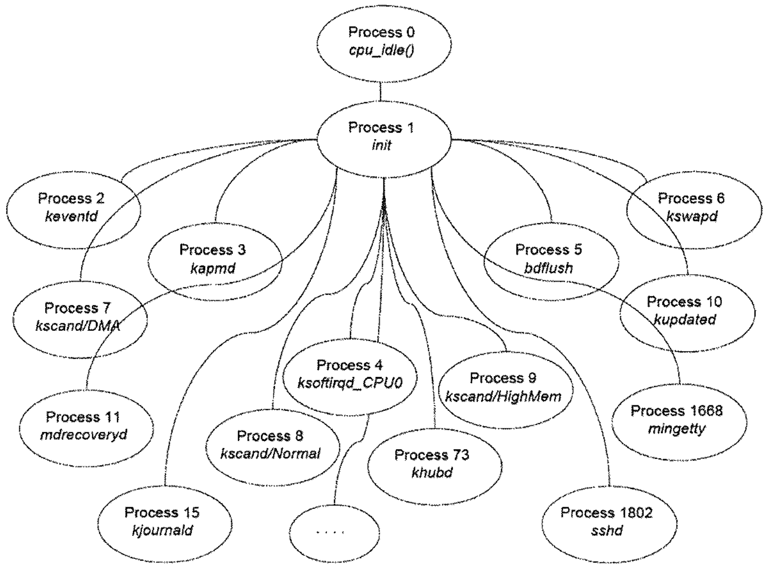

Figure 7. Process Hierarchy within Linux

Once *do_basic_setup()* returns, *do_initcalls()* is invoked to perform most of the low-level initialization of network and other types of devices, is invoked. This function calls a number of initialization functions specific to various devices and subsystems, as well as driver initialization routines. Any function wrapped with the macro *__initcall* or *module_init* will be invoked by *do_initcalls* at this point. The *__initcall* and *module_init* macros instruct the *gcc* complier to place the respective function in an ELF section named *.text.init* when linking, while the entry address of the function is stored in the *.initcall.init* section. Thus *do_initcalls()* involves a relatively simple loop that moves through the list of function addresses in *.initcall.init* and calls each of them. This unique structuring of the code allows a clean seven-line implementation of *do_initcall()*.

Linux makes extensive use of the ability of gcc to place portions of code into specific sections when linking object files, as provided by the *__attribute__* directive [14]. of the 27 sections defined in the Linux kernel contrasts sharply with the nine that are found in the OpenBSD kernel image. Using the *__attribute__* construction not only allows for the clean implementation of functions like *do_initcall()*, but also allows the operating system to use memory more efficiently. Since all functions marked with the *__init* macro are put into the *.text.init* section when compiled and linked, *free_initmem()* can free the memory pages taken up by that section after initialization is complete.

After all drivers have been initialized, *do_initcalls()* returns at which point its parent function *do_basic_setup()* to also return, placing control back inside of the *init()* function. The mount points of ramdisks and other devices, such as consoles, are determined next by invoking *prepare_namespace()*. At

[2] This thread previously was used to run the tasks that accumulated on the *tq_scheduler* queue, but that queue was deleted in an early version of the 2.4 kernel. See comments on *schedule_task()* in *kernel/context.c* for details.

[3] The fact that some groups of processes connect to Process 1 at the same point does not indicate any special relationship.

this point the system has completed the tasks essential to a working runtime environment. Now focus shifts to the completion of user space initialization tasks.

The function *free_initmem()* is called to release from kernel memory sections defined as necessary only at initialization: *.text.init, .data.init, .setup.init* and *.initcall.init*. As no more operations will be performed that are sensitive to timing and race conditions, this thread no longer needs to ensure it has exclusive access to kernel data structures. Therefore Process 1 is next able to release the kernel lock it obtained upon entry to the *init()* function, by calling *unlock_kernel()*. Also, to ensure it does not inadvertently continue to have access to kernel data, the thread releases files currently being shared.

Last, Process 1 in kernel mode opens the default console as standard input (file descriptor 0). The function *dup()* is then called twice on file descriptor 0 to allow the usage of the same console for standard output (file descriptor 1) and standard error (file descriptor 2). Finally *run_init_process()* is called, which invokes *execve()* on the argument it is passed. If this fails, it tries to execute, in order, */sbin/init*, */etc/init* and */bin/init*. As a last resort the kernel attempts to execute the */bin/sh* shell as the Process 1 binary, in an attempt to provide an interface for fixing the malfunctioning system.

To complete this discussion, the rest of Process 0 execution will be traced. After forking Process 1 while inside the function call *rest_init()*, Process 0 releases locked kernel resources, via a call to *unlock_kernel()*. It then sets a flag to indicate that the current process, i.e., itself Process 0, should be rescheduled. A call is then made to the function *cpu_idle()*. Once inside this function, control will not return. The function consists of a low-priority idle loop where Process 0 sits waiting for other processes to request rescheduling. When such a request is made, it is Process 0 that will call the scheduler.

With Process 0 in an idle loop and Process 1 executing the *init* binary, Linux kernel-level initialization is complete.

5 Conclusions

The initialization of both OpenBSD and Linux can be divided into two main phases, excluding the BIOS: the boot loader phase and the kernel initialization phase. While both operating systems must employ assembly language code to handle tasks like probing memory or setting up operating system controlled registers, a major difference between the two is where this assembly language code resides. OpenBSD is distributed with an integrated boot loader that moves the processor into protected memory mode. Utilizing a boot loader that handles these assembly level details allows the kernel entry point to begin with a much more robust environment, including memory protection. Therefore the amount of assembly code needed within the OpenBSD kernel has been minimized. On the other hand, the Linux kernel does not include any multi-stage boot loader code. It typically relies on the existence of third-party boot loader programs, and does not assume that the processor is put into protected memory mode. Instead the Linux kernel moves the processor into protected mode itself, requiring the kernel to include the assembly language code to establish protected memory mode. This means that, overall, the Linux kernel must execute a larger amount of assembly code than OpenBSD before reaching its high-level initialization function, *start_kernel()*.

The *start_kernel()* function in Linux largely resembles the *main()* routine of OpenBSD. Both are called after the CPU has been put into protected mode. Code that requires direct access to registers or flat memory addresses has already been executed. Both *start_kernel()* and *main()* are processor-independent and all code within these two functions is written strictly in C. Any machine-dependent functions are hidden within the functions called by these two high-level routines. Each subsystem within both operating systems defines an initialization function that is executed by the high-level initialization routine. In both OpenBSD and Linux these initialization functions are defined as part of the subsystems themselves, not as part of a separate initialization module.

While implementation details are slightly different, the overall architecture used to bring both operating systems up to a running state is essentially the same. Processor-specific items, e.g. moving the processor into protected mode on an Intel x86 CPU, are first handled by assembly code. Once these low-level details are sufficiently handled control moves to a high-level machine-independent initialization function. This function completes initialization, including the creation of any additional processes needed by the system.

References

[1] Dodge, C., *Recommendations for Secure Initialization Routines in Operating Systems*. Masters thesis, Naval Postgraduate School, 2004.

[2] Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model. Version 2.2, Revision 256. http://www.commoncriteriaportal.org/public/files/ccpart1v2.2.pdf. January 2004.

[3] LILO (The Linux Loader), http://freshmeat.net/projects/lilo/. Accessed May 2004.

[4] GRUB (Grand Unified Boot Loader), http://www.gnu.org/software/grub/. Accessed June 2004.

[5] Okuji, Y.K, Ford, B., Boleyn, E.S., Ishiguro, K., *The Multiboot Specification*, Free Software Foundation, 2002. Available from http://www.gnu.org/software/grub/manual/multiboot/multiboot.pdf. Accessed May 2004.

[6] Mach operating system project, http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html. Accessed November 2004.

[7] Fiasco microkernel, http://os.inf.tu-dresden.de/fiasco/. Accessed October 2004.

[8] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, http://www.intel.com/design/pentium4/manuals/index_new.htm. Accessed May 2004.

[9] OpenBSD Security, http://www.openbsd.org/security.html. Accessed June 2004.

[10] Intel Corporation, Nov. 20 1997. "Invalid Instruction Erratum Overview" Available from http://support.intel.com/support/processors/pentium/ppiie/. Accessed September 2004.

[11] Bovet, D. P. and Cesati, M., *Understanding the Linux Kernel, 2nd edition*, O'Reilly & Associates, 2002.

[12] Tool Interface Standard (TIS) Committee, *Executable and Linking Format (ELF) Specification*, Version 1.2, http://www.x86.org/ftp/manuals/tools/elf.pdf. Accessed August 2004.

[13] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, http://www.intel.com/design/pentium4/manuals/index_new.htm. Accessed May 2004.

[14] Free Software Foundation, *GCC 3.4.1 Manual*, http://gcc.gnu.org/onlinedocs/gcc-3.4.1/gcc/. Accessed July 2004.